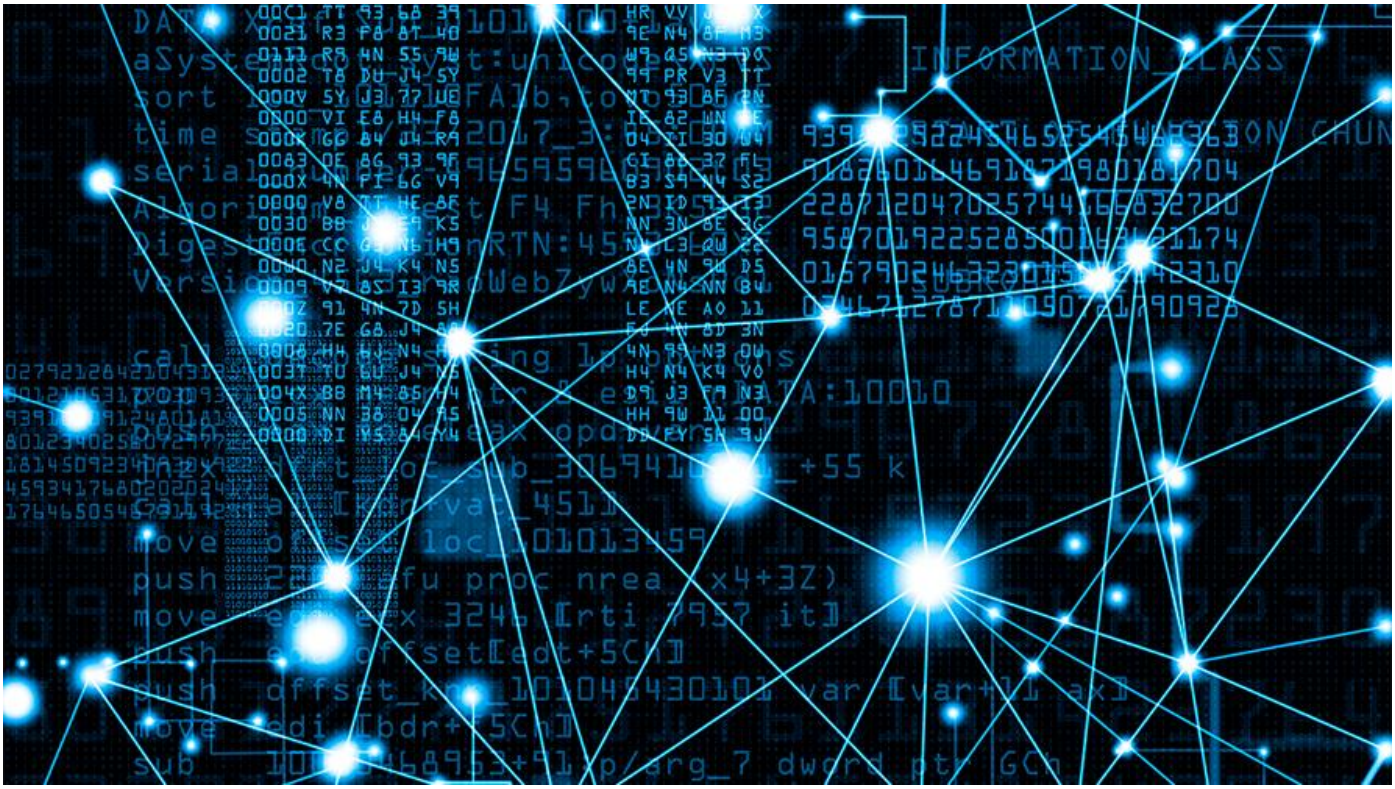
ΑΛΓΟΡΙΘΜΟΙ ΚΑΙ ΠΟΛΥΠΛΟΚΟΤΗΤΑ

1Η ΣΕΙΡΑ ΓΡΑΠΤΩΝ ΑΣΚΗΣΕΩΝ



NOVEMBER 21, 2021

ΘΟΔΩΡΗΣ ΑΡΑΠΗΣ – EL18028

Άσκηση 1

1.

$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$

$n^{\log_b a} = n^{\log_2 4} = n^2$

$f(n) = \Theta(n^2 \log n) = n^2 \log n$

Δεν είναι πολυωνυμικά διαχωρίσιμες, οπότε δεν εφαρμόζεται το Master Theorem. Θα το λύσουμε επαγωγικά:

$T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n = 4\left(4T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2 \log\left(\frac{n}{2}\right)\right) + n^2 \log n \Longrightarrow$

$T(n) = 4^2 T\left(\frac{n}{2^2}\right) + n^2 \log\left(\frac{n}{2}\right) + n^2 \log n = 4^2 T\left(\frac{n}{2^2}\right) + n^2 \log\left(\frac{n}{2}\right) + n^2 \log n \Longrightarrow$

$T(n) = 4^3 T\left(\frac{n}{2^3}\right) + n^2 \log\left(\frac{n}{2^2}\right) + n^2 \log\left(\frac{n}{2}\right) + n^2 \log n \Longrightarrow$

. . .

$T(n) = 4^k T\left(\frac{n}{2^k}\right) + n^2 \log\left(\frac{n}{2^{k-1}}\right) + n^2 \log\left(\frac{n}{2^{k-2}}\right) + \cdots + n^2 \log n \Longrightarrow$

Ο όρος $4^k T\left(\frac{n}{2^k}\right)$, για $k = \log n$, μας δίνει: $4^{\log n} T(1) = n^{\log 4}\Theta(1) = n^2$

Οι υπόλοιποι όροι μας δίνουν:

$n^2 \log\left(\frac{n}{2^{k-1}}\right) + n^2 \log\left(\frac{n}{2^{k-2}}\right) + \cdots + n^2 \log n =$

$= n^2\left(\log\left(\frac{n}{2^{k-1}}\right) + \log\left(\frac{n}{2^{k-2}}\right) + \cdots + \log n\right) = n^2 \log\left(\frac{n^{k-1}}{1 \cdot 2 \cdot \ldots \cdot \frac{n}{2}}\right)$

Από το οποίο προκύπτει εν τέλει $\Theta(n^2 \log^2 n)$. Συνεπώς, θα έχουμε:

$$T(n) = \Theta(n^2) + \Theta(n^2 \log^2 n) \Longrightarrow$$

$$\boxed{T(n) = \Theta(n^2 \log^2 n)}$$

2.

$T(n) = 5T\left(\frac{n}{2}\right) + \Theta(n^2 \log n)$

$n^{\log_b a} = n^{\log_2 5} = n^{2.3219}$

$f(n) = \Theta(n^2 \log n) = n^2 \log n$

Το $f(n)$ είναι γραμμικά μικρότερο από το $n^{\log_b a}$, αφού

$\frac{n^{2.3219}}{n^2 \log n} > n^{\varepsilon} \Rightarrow n^{0.3219-\varepsilon} > \log n\,, \qquad \varepsilon \in (0, 0.3219)$

οπότε εφαρμόζεται το Master Theorem. Θα ισχύει:

$T(n) = \Theta\left(n^{\log_b a}\right)$ αφού $f(n) = O(n^{\log_b a - \varepsilon})$

$$\boxed{T(n) = \Theta(n^{2.3219})}$$

3.

$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + \Theta(n)$

Αποτελεί ειδική περίπτωση του Master Theorem και ισχύει:

$\gamma_1 = \frac{1}{4}\,, \gamma_2 = \frac{1}{2}$

Οπότε ισχύει:

$\gamma_1 + \gamma_2 = \frac{1}{4} + \frac{1}{2} = \frac{3}{4} < 1 - \varepsilon, \qquad \varepsilon > 0\ (3\eta\ \pi\varepsilon\rho\acute{\iota}\pi\tau\omega\sigma\eta)$

Και άρα:

$$\boxed{T(n) = \Theta(n)}$$

4.

$$T(n) = 2T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + \Theta(n) = T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{2}\right) + \Theta(n)$$

Αποτελεί ειδική περίπτωση του Master Theorem και ισχύει:

$$\gamma_1 = \frac{1}{4}\,, \gamma_2 = \frac{1}{4}\,, \gamma_3 = \frac{1}{2}$$

Οπότε ισχύει:

$$\gamma_1 + \gamma_2 + \gamma_3 = \frac{1}{4} + \frac{1}{4} + \frac{1}{2} = \frac{3}{4} = 1\ (2\eta\ \pi\varepsilon\rho\acute{\iota}\pi\tau\omega\sigma\eta)$$

Και άρα:

$$\boxed{T(n) = \Theta(n \log n)}$$

5.

$$T(n) = T\left(n^{\frac{1}{2}}\right) + \Theta(\log n)$$

Θα έχουμε επαγωγικά:

$$T(n) = T\left(n^{\frac{1}{2}}\right) + \log n = T\left(n^{\frac{1}{2^2}}\right) + \log n\left(1 + \frac{1}{2}\right) = T\left(n^{\frac{1}{2^3}}\right) + \log n\left(1 + \frac{1}{2} + \frac{1}{4}\right) \Longrightarrow$$

$$T(n) = T\left(n^{\frac{1}{2^k}}\right) + \log n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{k-1}}\right)$$

Για $k \to \infty$ θα ισχύει:

$$\lim_{k\to\infty} n^{\frac{1}{2^k}} = 1 \quad \kappa\alpha\iota \quad \sum_{k=1}^{\infty}\left(\frac{1}{2}\right)^k = 1$$

Άρα ο πρώτος όρος του $T(n)$ δίνει $T(1) = \Theta(1)$ και ο δεύτερος $\Theta(\log n)$

Οπότε: $T(n) = \Theta(1) + \Theta(\log n)$ και επομένως:

$$\boxed{T(n) = \Theta(\log n)}$$

6.

$$T(n) = T\left(\frac{n}{4}\right) + \Theta(\sqrt{n}) = T\left(\frac{n}{4}\right) + \Theta\left(n^{\frac{1}{2}}\right)$$

$$\log_b a = \log_4 1 = 1$$

$$d = \frac{1}{2} > 0 = \log_b a$$

Οπότε, ως ειδική περίπτωση του Master Theorem θα ισχύει:

$$T(n) = \Theta(n^d) = \Theta\left(n^{\frac{1}{2}}\right) \Longrightarrow$$

$$\boxed{T(n) = \Theta(\sqrt{n})}$$

Άσκηση 2

Υποθέτουμε δεικτοδότηση πίνακα A[1], A[2], …, A[*n*]

(α) Τα βήματα του αλγορίθμου είναι τα εξής:

- Διατρέχουμε τον πίνακα Α μέχρι να βρούμε το μεγαλύτερο στοιχείο. (αποθηκεύουμε τον δείκτη του, έστω *k*)
- Εκτελούμε μία προθεματική περιστροφή για τα *k* πρώτα στοιχεία.
- Τώρα το μεγαλύτερο στοιχείο βρίσκεται στην πρώτη θέση του πίνακα, οπότε εκτελούμε μία ακόμα προθεματική περιστροφή σε όλα τα στοιχεία του πίνακα, με αποτέλεσμα το μεγαλύτερο στοιχείο να βρεθεί στο τέλος του πίνακα.
- Στην συνέχεια, τρέχουμε τον αλγόριθμο αναδρομικά για τον πίνακα A[1..*n*-1]
- Όταν φτάσουμε να εφαρμόσουμε τον αλγόριθμο για τον πίνακα A[1..3], παρατηρούμε ότι χρειάζονται το πολύ 3 περιστροφές για να ταξινομηθεί ο πίνακας, σε κάθε περίπτωση (το πολύ 2 περιστροφές για να τοποθετηθεί σωστά το τρίτο στοιχείο και το πολύ άλλη μία για τα δύο πρώτα). Οπότε σταματάμε την αναδρομή στο σημείο αυτό και χρησιμοποιούμαι τις ελάχιστες απαραίτητες περιστροφές.
- Extra: εφόσον δεν έχουμε duplicates, μπορούμε να μειώσουμε τις περιστροφές σε ορισμένες περιπτώσεις αν πριν από κάθε περιστροφή ελέγχουμε αν το μεγαλύτερο στοιχείο βρίσκεται στην σωστή του θέση (συνθήκη: A[*k*] = *k*).

Ο αλγόριθμος θα εκτελέσει το πολύ 2*n*-3 περιστροφές.

(β) Ακολουθούμε τα ίδια βήματα με προηγουμένως, αναζητώντας τώρα το μεγαλύτερο στοιχείο κατά απόλυτη τιμή. Προσθέτουμε ακόμη μία συνθήκη στην αναδρομή:

- Αφού φέρουμε με μία περιστροφή το μεγαλύτερο στοιχείο στην αρχή του πίνακα, ελέγχουμε το πρόσημό του. Αν είναι θετικό, τότε εκτελούμε μία περιστροφή στο πρώτο μόνο στοιχείο, προκειμένου να αλλάξει πρόσημο και συνεχίζουμε κανονικά την εκτέλεση του αλγορίθμου. Αν είναι αρνητικό, συνεχίζουμε απλά την εκτέλεση του αλγορίθμου.

Ο αλγόριθμος θα εκτελέσει λιγότερες προθεματικές περιστροφές από 3*n*.

(γ)

1. Σε κάθε πίνακα A_t (που δεν είναι ο [-1, -2, …, -*k*] ή ο [*k*, …, 2, 1], η λύση σε αυτές τις περιπτώσεις περιγράφεται στο ερώτημα 2.(γ)) θα ισχύει τουλάχιστον μία από τις παρακάτω περιπτώσεις (Αγνοώντας τα ήδη υπάρχοντα συμβατά ζεύγη):

    - Βρίσκουμε το μεγαλύτερο στοιχείο κατά απόλυτη τιμή στον πίνακα A_t. Αν το στοιχείο αυτό έχει θετικό πρόσημο τότε με 2 περιστροφές μπορούμε να το πάμε στην τελευταία θέση (διατηρώντας έτσι το θετικό του πρόσημο) και να δημιουργήσουμε ένα συμβατό ζεύγος. Αν είναι αρνητικό και βρίσκεται στην πρώτη θέση τότε με 1 περιστροφή το πάμε στο τέλος και φτιάχνουμε πάλι συμβατό ζεύγος.
    - Αν ένα στοιχείο (έστω *k*) έχει θετικό πρόσημο και βρίσκεται αριστερότερα του στοιχείου $x = k + 1$ (θετικό), τότε με 2 περιστροφές μπορούμε να δημιουργήσουμε συμβατό ζεύγος, μία περιστροφή για να πάμε το *k* στην αρχή (αν είναι ήδη στην αρχή τότε εκτελούμε απλά μία περιστροφή μόνο σε αυτό για να αλλάξει πρόσημο) και άλλη μία για να το φέρουμε αριστερά του *x*. Αν το *k* είναι αρνητικό και βρίσκεται στην πρώτη θέση τότε με 1 περιστροφή το φέρνουμε αριστερά του *x* (όπου $x = |k| + 1$).
    - Αν ένα στοιχείο (έστω *k*) έχει αρνητικό πρόσημο και βρίσκεται αριστερότερα του στοιχείου *x* με απόλυτη τιμή $|x| = |k| - 1$ το οποίο έχει και αυτό αρνητικό πρόσημο, τότε με 2 περιστροφές μπορούμε να δημιουργήσουμε συμβατό ζεύγος μία περιστροφή για να πάμε το *k* στην αρχή (αν είναι ήδη στην αρχή τότε εκτελούμε απλά μία περιστροφή μόνο σε αυτό για να αλλάξει πρόσημο) και άλλη μία για να το φέρουμε αριστερά του *x*. Αν το *k* είναι θετικό και βρίσκεται στην πρώτη θέση τότε με 1 περιστροφή το φέρνουμε αριστερά του *x*.
    - Αν ένα στοιχείο (έστω *k*) έχει θετικό πρόσημο και βρίσκεται δεξιότερα του στοιχείου $x = -(k + 1)$ (αρνητικό), τότε με 2 περιστροφές μπορούμε να δημιουργήσουμε συμβατό ζεύγος, μία περιστροφή για να πάμε το *k* στην αρχή και άλλη μία για να το φέρουμε αριστερά του *x*.

- Αν ένα στοιχείο (έστω k) έχει αρνητικό πρόσημο και βρίσκεται δεξιότερα του στοιχείου $x = |k| - 1$ (θετικό), τότε με 2 περιστροφές μπορούμε να δημιουργήσουμε συμβατό ζεύγος, μία περιστροφή για να πάμε το k στην αρχή και άλλη μία για να το φέρουμε αριστερά του x.

2. Με βάση την παραπάνω παρατήρηση, μπορούμε να κατασκευάσουμε τον εξής αλγόριθμο:

Όσο το μήκος του πίνακα είναι μεγαλύτερο του 1:

- Ελέγχουμε αν υπάρχουν συμβατά ζεύγη. Αν υπάρχουν, αντικαθιστούμε τα ζεύγη με ένα tuple της μορφής $(x, x + 1)$. (Κάθε φορά που δημιουργείται ένα tuple μειώνεται το μέγεθος του πίνακα κατά 1).
- Εξετάζουμε αν ο πίνακας είναι της μορφής $[-1, -2, \ldots, -k]$ ή της μορφής $[k, \ldots, 2, 1]$. (Τα tuples θα ελέγχονται στην συνθήκη σαν να ήταν προέκταση του πίνακα, θα θεωρούνται όμως ως ένα στοιχείο του πίνακα)
  Αν είναι της μορφής $[-1, -2, \ldots, -k]$ τότε για k φορές επαναλαμβάνουμε τα εξής:
  - Εκτελούμε μία προθεματική περιστροφή και των k στοιχείων.
  - Εκτελούμε μία προθεματική περιστροφή των $k - 1$ στοιχείων.

  Μετά από k επαναλήψεις ο πίνακας θα έχει ταξινομηθεί. (**$2k$** περιστροφές το πολύ).

  Αν είναι της μορφής $[k, \ldots, 2, 1]$ τότε κινούμαστε με τον ίδιο τρόπο αλλά εκτελούμε πρώτη την περιστροφή των $k - 1$ στοιχείων και μετά των k. Επιπλέον στην τελευταία επανάληψη δεν εκτελούμε την προθεματική αλλαγή των k στοιχείων. (**$2k$-1** περιστροφές το πολύ)

  Αλλιώς:

  Δημιουργούμε κάποιο συμβατό ζεύγος (ερώτημα 2.(β)). Τα tuples συγκρίνονται με το πρώτο τους στοιχείο για να δημιουργήσουν ζεύγος από αριστερά, και με το τελευταίο στοιχείο για να δημιουργήσουν ζεύγος από δεξιά. Οπότε θα δημιουργούνται στοιχεία της μορφής $(\ldots, x - 1, x, x + 1, x + 2, \ldots)$. Η αντιστροφή ενός tuple θα γίνεται ως εξής: $(x, y, \ldots, z) \rightarrow (-z, \ldots, -y, -x)$.

  (Επαναλαμβάνουμε τα παραπάνω βήματα του αλγορίθμου μέχρι το μέγεθος του πίνακα να ισούται με 1). (**$2n$** περιστροφές το πολύ).

Στο τέλος θα έχουμε έναν πίνακα 1x1 με ένα tuple με τα στοιχεία ταξινομημένα. Οπότε, μετατρέπουμε το tuple σε πίνακα και έχουμε το ζητούμενο.

Ο αλγόριθμος θα εκτελέσει στη χειρότερη περίπτωση το πολύ $2n$ προθεματικές περιστροφές.

Άσκηση 3

Υποθέτουμε δεικτοδότηση πίνακα $A[0],\ A[1], \dots ,\ A[n-1]$

Ο αλγόριθμος είναι ο εξής:

Δημιουργούμε ένα stack στο οποίο θα αποθηκεύουμε tuples της μορφής $(i,\ A[i])$, αρχικοποιημένο με την τιμή $(0, \infty)$.

Μπορούμε να αποθηκεύουμε το αποτέλεσμα σε έναν πίνακα $Result[n]$ με την μορφή $Result[i] = j$, όπου j η θέση που κυριαρχεί της i. Τον αρχικοποιούμε με την τιμή μηδέν σε όλες τις θέσεις.

Αρχικοποιούμε το $i = 2$

Όσο ισχύει $i < n$ επανέλαβε:

- Αν ισχύει $A[i] > A[i-1]$ τότε επανέλαβε:
  - Κάνε pop ένα στοιχείο από το stack (Έστω $k[2] = (j, A[j])$). Σύγκρινε το $A[j]$ με το $A[i]$.
    Αν ισχύει $A[j] > A[i]$ τότε θέσε $Result[i] = j$, ξαναβάλε το στοιχείο k στο stack, αύξησε την τιμή του i κατά 1 και βγες από το loop.
    Αλλιώς επανέλαβε.
- Αλλιώς θέσε $Result[i] = i - 1$, πρόσθεσε το $(i-1,\ A[i-1])$ στο stack και αύξησε την τιμή του i κατά 1.

Εξήγηση:

Ο αλγόριθμος θα ελέγχει πρώτα αν το $A[i]$ είναι μεγαλύτερο από το $A[i-1]$. Αν ναι, τότε βγάζουμε ένα-ένα στοιχείο από το stack μέχρι να πετύχουμε κάποιο στοιχείο $k[2] = (j, A[j])$ τέτοιο ώστε $A[j] > A[i]$. Ύστερα ξαναβάζουμε το k στο stack, ενημερώνουμε τον $Result[n]$ και αυξάνουμε το i κατά ένα. Έτσι, αφαιρούμε από το stack όλα τα μικρότερα στοιχεία που προηγούνται του $A[i]$. Επομένως, αποφεύγουμε τις άσκοπες συγκρίσεις στο μέλλον και το stack έχει την εξής μορφή: κάτω από κάθε στοιχείο $(i,\ A[i])$ βρίσκεται το κυρίαρχο στοιχείο του (Συγκεκριμένα το ζεύγος $(j,\ A[j])$, όπου το j είναι η κυρίαρχη θέση του i.

Στην περίπτωση που το $A[i]$ είναι μικρότερο από το $A[i-1]$, τότε ενημερώνουμε τον $Result[n]$ και προσθέτουμε το $(i-1,\ A[i-1])$ στο stack θέτουμε $i = i + 1$.

Ο ισχυρισμός που μας καθοδηγεί είναι πως αν το εξεταζόμενο στοιχείο του πίνακα είναι μεγαλύτερο από το στοιχείο της κυρίαρχης θέσης του προηγούμενου στοιχείου του, τότε η επόμενη πιθανή κυρίαρχη θέση του εξεταζόμενου στοιχείου είναι η κυρίαρχη θέση της κυρίαρχης θέσεις του προηγούμενού του στοιχείου (καθώς όλα τα ενδιάμεσα στοιχεία θα είναι μικρότερα από το εξεταζόμενο).

Η πολυπλοκότητα του αλγορίθμου θα είναι $\boldsymbol{O(n)}$, καθώς θα κάνει n επαναλήψεις και συνολικά το πολύ $2n-1$ συγκρίσεις (μιας και αφαιρούμε τα μικρότερα στοιχεία από το stack).

Άσκηση 4

Ο αλγόριθμος θα ακολουθεί τα εξής βήματα:

Έχουμε τον πίνακα $a[1..n]$ (ταξινομημένος και με duplicates) με τους χρόνους άφιξης και d την καθυστέρηση κάθε αυτοκινήτου.

Αρχικοποιούμε τις μεταβλητές:

$time = a[0]$ $(\theta\alpha\ \mu\epsilon\tau\rho\acute{\alpha}\epsilon\iota\ \tau\eta\nu\ \chi\rho o\nu\iota\kappa\acute{\eta}\ \sigma\tau\iota\gamma\mu\acute{\eta}\ \pi o\upsilon\ \beta\rho\iota\sigma\kappa\acute{o}\mu\alpha\sigma\tau\epsilon)$,

$plugged = 1$ $(\pi\lambda\acute{\eta}\theta o\varsigma\ \tau\omega\nu\ occupied\ \pi\rho\iota\zeta\acute{\omega}\nu)$,

$i = 1$ $(iterator)$

$flag = False$ $(boolean\ \gamma\iota\alpha\ \acute{\epsilon}\lambda\epsilon\gamma\chi o\ \sigma\upsilon\nu\theta\acute{\eta}\kappa\eta\varsigma)$

και ένα queue με όνομα $wait$ που θα αποθηκεύει τα αυτοκίνητα που περιμένουν στον σταθμό.

Αρχικά βρίσκουμε το μεγαλύτερο πλήθος duplicates (σε $O(n)$) διατρέχοντας τον πίνακα και μετρώντας τον αριθμό των duplicates ενός στοιχείου που εμφανίζονται, κρατώντας στο τέλος το μεγαλύτερο από όλους τους αριθμούς αυτούς. Σε μορφή ψευδοκώδικα:

$s = 1, current = a[0], count = 1$

$\Gamma\iota\alpha\ j\ \alpha\pi\acute{o}\ 0\ \acute{\epsilon}\omega\varsigma\ n\ \epsilon\pi\alpha\nu\acute{\epsilon}\lambda\alpha\beta\epsilon$:

$A\nu\ a[j] = current\ \tau\acute{o}\tau\epsilon$

$count = count + 1$

$A\lambda\lambda\iota\acute{\omega}\varsigma$

$current = a[j]$

$s = \max(count, s)$

$count = 1$

Τώρα γνωρίζουμε ότι το μέγιστο επιτρεπτό πλήθος πριζών είναι s. Θα εφαρμόσουμε binary search στο πλήθος των πριζών για να επιταχύνουμε την αναζήτηση. Οπότε θα έχουμε:

$s^* = \frac{s+1}{2}\ (ceil)$

$Θέτουμε\ low = 0, high = s$

Αναζητάμε το ελάχιστο πλήθος πριζών ως εξής (ως ψευδοκώδικα):

$Όσο\ high > low + 1\ επανέλαβε$:

- $Όσο\ i < n\ επανέλαβε$:

  $Αν\ time = a[i]\ και\ plugged < s^*\ τότε$

  $plugged = plugged + 1$

  $i = i + 1$

  $Αλλιώς\ αν\ time = a[i]\ τότε$

  $Βάλε\ στην\ wait\ το\ a[i]$

  $i = i + 1$

  $Αλλιώς$ (Περίπτωση $time < a[i]$)

  $time = time + 1$

  $plugged = 0$

  $Όσο\ plugged < s^*\ και\ η\ wait\ δεν\ είναι\ κενή\ επανέλαβε$:

  $Κάνε\ pop\ ένα\ στοιχείο\ απο\ την\ wait\ (έστω\ k)$

  $Αν\ time - k \geq d - 1\ τότε$

  $plugged = plugged\ + 1$

  $Αλλιώς$

  $flag = True$

  $τερμάτισε\ τα\ δύο\ loops$

- $Αν\ flag = True\ τότε\ θέσε\ low = s^*, s^* = \frac{low+high}{2}(ceil), flag = false$
  $και\ επανέλαβε\ την\ παραπάνω\ διαδικασία$
- $Αλλιώς\ θέσε\ result = s^*, high = s^*,\ s^* = \frac{low+high}{2}\ (ceil)$
  $και\ επανέλαβε\ την\ παραπάνω\ διαδικασία$

Ο αλγόριθμος θα τερματίσει όταν $low + 1 = high$. Τότε θα έχουμε βρει το ελάχιστο s^* ($μεταβλητή\ result$).

Εξήγηση:

Ο αλγόριθμος θα εξετάζει κάθε s^* αν ικανοποιεί την προϋπόθεση του προβλήματος για d καθυστέρηση. Συγκεκριμένα θα εκτελεί $2n$ το πολύ επαναλήψεις για κάθε s^*. Κάθε αυτοκίνητο που φτάνει θα μπαίνει για φόρτιση σε κάποια κενή πρίζα. Μετά από 1 χρονική μονάδα θα αδειάζονται όλες οι πιασμένες πρίζες. Αν κατά την άφιξη ενός αμαξιού δεν υπάρχει κενή πρίζα, τότε το αμάξι (η χρονική στιγμή άφιξής του) τοποθετείται σε μία ουρά ωσότου ελευθερωθεί κάποια πρίζα. Η συνθήκη $time - k \geq d - 1$ ελέγχει αν κάποιο αυτοκίνητο έχει ξεπεράσει την καθυστέρηση d. Η πολυπλοκότητα του αλγορίθμου είναι $\boldsymbol{O(n \log n)}$. (Το $\log n$ προκύπτει από το binary search)

Άσκηση 5

(α)

Θα λύσουμε το πρόβλημα με τη μέθοδο του binary search για το σύνολο $[0, M]$.

Έχουμε ως δεδομένη την $F_S(\ell) = |\{x \in S : x \leq \ell\}|$ και τον φυσικό k.

Τα βήματα του αλγορίθμου είναι τα εξής:

- $low = 0, high = M$
- $Όσο\ low < high - 1\ επανέλαβε$:

  $x = \dfrac{low + high}{2}\ (floor)$

  $Αν\ F_S(x) < k\ τότε$

  $low = x$

  $Αλλιώς\ αν\ F_S(x) > k\ τότε$

  $high = x$

  $Αλλιώς\ έξοδος\ από\ το\ loop$ (βρήκαμε την λύση)

Μόλις βγει από το loop η ζητούμενη λύση θα είναι το x. Η πολυπλοκότητα του αλγορίθμου θα είναι $\boldsymbol{O(\log M)}$.

(β)

Έχουμε τον πίνακα $A[1 \ldots n]$ με μέγιστο στοιχείο το M. Επιπλέον, κάθε στοιχείο στον πίνακα Α είναι μοναδικό. Σύμφωνα με αυτά τα δεδομένα υπολογίζουμε ότι το πολυσύνολο S θα έχει $\frac{n(n-1)}{2}$ στοιχεία. Συνεπώς οι γραμμικοί αλγόριθμοι στο πολυσύνολο θα είχαν στην καλύτερη $O(n^2)$ πολυπλοκότητα. Οπότε επιλέγουμε να δουλέψουμε στον πίνακα $A[1 \ldots n]$. Θα υλοποιήσουμε την συνάρτηση $F_S(\ell) = |\{x \in S : x \leq \ell\}|$.

Ο αλγόριθμος θα ακολουθήσει τα εξής βήματα:

- Ταξινομούμε τον πίνακα *A*. Υπάρχουν διάφοροι αλγόριθμοι ταξινόμησης, επιλέγουμε όμως τον Merge Sort για να ταξινομήσουμε τον πίνακα με την ελάχιστη δυνατή πολυπλοκότητα. ($\boldsymbol{O(n \log n)}$)
- Διατρέχουμε τον A_{sorted} και δημιουργούμε έναν πίνακα $diff[n-1]$ που περιέχει τις ανά δύο διαφορές των στοιχείων του A_{sorted}. Ποιο συγκεκριμένα:

$$diff[i] = A_{sorted}[i+1] - A_{sorted}[i], i \in [0, n-2]$$
$$\boldsymbol{O(n-1)}$$

- Από τον πίνακα $diff[n-1]$ μπορούμε να βρούμε το πλήθος των στοιχείων του S που είναι μικρότερα από έναν θετικό αριθμό ℓ σε γραμμικό χρόνο. Αυτό συμβαίνει διότι η μορφή του πίνακα μας επιτρέπει να εξετάσουμε όλες τις διαφορές των στοιχείων του πίνακα A_{sorted} διατρέχοντας τον πίνακα $diff[n-1]$ με δύο δείκτες και μετρώντας τους πιθανούς συνδυασμούς αθροισμάτων διαδοχικών στοιχείων του που είναι μικρότεροι του ℓ.
  Βασικός ισχυρισμός: οι πιθανοί συνδυασμοί διαδοχικών αθροισμάτων ενός ακριανού στοιχείου με (και χωρίς) τα υπόλοιπα στοιχεία του πίνακα είναι τόσοι όσο και το μήκος του πίνακα.
  Ακολουθεί ο ψευδοκώδικας $\big((x \pm= \cdots) \rightleftharpoons (x = x \pm \cdots)\big)$:
  - $Αρχικοποίησε\ τις\ μεταβλητές$
    $count = 0, sum = diff[0], i = 0, j = 1, flag = True$
  - $Αν\ diff[0] \leq \ell\ τότε$
    $θέσε\ count = 1$
  - Όσο $i \neq n-2$ ή $j \neq n-2$ επανέλαβε:
    - $Αν\ sum \leq \ell\ και\ j < n-2\ τότε$
      $θέσε\ j \mathrel{+}= 1, sum \mathrel{+}= diff[j]$
    - $Αλλιώς$
      $θέσε\ sum \mathrel{-}= diff[i], i \mathrel{+}= 1$
    - $Αν\ sum \leq \ell\ \ και\ flag = True\ τότε$
      $θέσε\ count \mathrel{+}= j - i + 1$
      $Αν\ j = n-2\ τότε$
      $θέσε\ flag = False$

  Ο παραπάνω αλγόριθμος θα έχει πολυπλοκότητα $\boldsymbol{O(2n-3)}$ αφού σε κάθε επανάληψη αυξάνουμε έναν δείκτη. (j από 1 έως $n-2$ και i από 0 έως $n-2$)

Συνεπώς, έχοντας υλοποιήσει παραπάνω την $F_S(\ell)$ με πολυπλοκότητα

$$O(n \log n + n - 1 + 2n - 3) = \boldsymbol{O(n \log n)}$$

μπορούμε να εφαρμόσουμε τον αλγόριθμο του ερωτήματος 5.(α). ($+\boldsymbol{O(\log M)}$)

Ο συνολικός αλγόριθμος θα έχει πολυπλοκότητα $\boldsymbol{O(n \log n + \log M)}$.